Bridge Design Pattern

1. Bridge – Introduction:

Now let's learn about Bridge design pattern. This is the Structural design pattern and it is one of those design pattern that may take some time for you to understand what exactly this does, because it promises something which may sound completely impossible when you first hear it. So you might have noticed that typically when we use inheritance, our abstraction and our implementation are tightly coupled. Of course, because our implementation is going to extend or implement our abstraction depending upon whether it is an abstract class or general(non-abstract) Java class or an interface. Now this Bridge design pattern promises that it can decouple our implementation and abstraction so that we can change any one of those without affecting the another. So we can add an operation/behaviour in our abstraction and we don't have to change our implementation. So how is that possible? Well, this design pattern achieves this feat by creating two separate inheritance hierarchies, and it may sound a little bit confusing, so please bear with me. We are going to learn what this means. But what this design pattern does is it creates two separate inheritance hierarchies. We have a inheritance hierarchy for abstraction. So abstraction is what our client code uses. So our client is going to use any one of the classes from our abstraction, and we have a separate hierarchy of implementation. And these two hierarchies are connected using composition. So when you read about it or hear about it, it sounds quite impossible to do. And that's because you may be thinking about having a change in the interface without needing change in its implementing class. That's not what this design pattern does. When it is talking about abstraction, it is talking about the interface that is used by client. When it is talking about implementation, it is talking about class that can provide that functionality. You can think about Adapter but not quite Adapter.

A screenshot of a computer

Description automatically generated

So let's look at the UML diagram, the formal UML diagram of this design pattern. To begin with, we have an Abstraction here, so this can be a base abstract class or interface that defines the interface(this interface means what is exposed to the client, for example, methods) that is used by our client code. Now this Abstraction also has a reference, as an instance variable, to whoever is providing that functionality. Next, we have a RefinedAbstraction, so RefinedAbstraction(it is a non-abstract class) is basically a child class(i.e., implementation) of our Abstraction, and it defines any specialized operation that we want. Then we have a Implementor. So Implementor is basically a base interface or a base class(it could be abstract or non-abstract) that provides methods that implement the functionality which is expected by our abstractor(i.e., Abstraction class). And what happens here is our Implementor can provide implementation in smaller steps. An example could be your Abstraction can define a method called as transfer money from account X to account Y. Your Implementor will have methods that says check the balance of account X, another method that checks whether account Y is active or whether it is a dormant account, it can have another method that actually takes the amount to be transferred from one account and deposit into second account. So the point here is that Implementor won't have 1 to 1 matching methods that are defined in Abstraction. So this class or interface(i.e., Implementor) is completely on its own, which is defining some methods and using some combination of these methods, we can implement the method/function that is defined in our Abstraction. Don't worry, this is going to get clearer as we learn the actual hands-on example. To move on, we have a ConcreteImplementorA and ConcreteImplementorB which implements whatever is defined in the Implementor abstract class/interface, and it provides the actual code that carries out that particular functionality. So this is the UML diagram for Bridge design pattern.

The name Bridge comes from this particular relationship(i.e., relationship between Abstraction and Implementor), the composition that is present between Abstraction and Implementor. So instance of Implementor is present inside of our Abstraction, and these are the two inheritance hierarchies. On the left side, we have an abstraction hierarchy, and on the right side we have an implementation hierarchy. And these two are connected by the composition relationship. This is what the Bridge refers to.

A screenshot of a computer

Description automatically generated

A diagram of a process

Description automatically generated with medium confidence

1. Bridge – Implementation Steps:

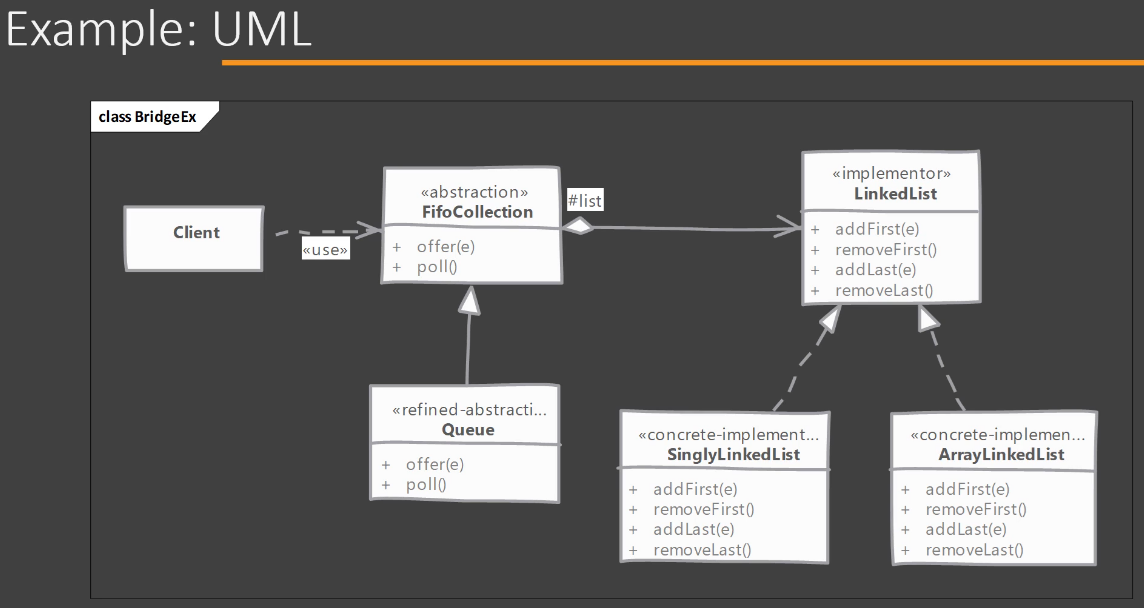
Now let's look at the steps that are needed in order to implement the Bridge design pattern. So we start by defining our Abstraction. So Abstraction is what our client code uses. So we are going to have methods in that particular base class that are used by our client. We can optionally also create a child class for this Abstraction that we define, and define more specialized operations on it. But that's completely an optional step. You can have one single class, which is an Abstraction. Now this single class doesn't have to be abstract. That's the important point and that is the key to understanding this design pattern. When we are talking about abstraction, we are not talking about abstract class or interface. We are talking about methods which abstract out the complexity or the implementation from the client(i.e., Abstraction is the client in this case). So Abstraction is not abstract class or interface, it is simply a method that hides what goes on while carrying out that functionality. That is what abstraction is. So it could be a regular class and we are still going to call that an abstraction because it hides how that operation is carried out. Next, we define our Implementor. So Implementor is again a separate class, so we can define it as an interface or abstract class. And in that particular Implementor, we are going to define methods which do not correspond to whatever is defined in our Abstraction. So again, taking the example of bank transfer. Our Abstraction class can define a method, single method that says transfer amount between these two accounts. So it will have a method called as transfer amount, which will take two account numbers and an amount. Implementor doesn't have to define/implement that method as it is. It can define methods which carry out smaller functionality. So for example, it can have a method that checks whether account is active or not, it can have a method that simply checks whether account has a given balance, third method that will check whether two accounts can transfer money to each other(whether there are any regulations or whether there are any problems with the transfer), and lastly, a method that simply carries out the transaction. So here in the Implementor, we have these smaller steps that we can carry out, and our Abstraction uses these smaller methods in order to provide the functionality. The object of Abstraction class is created by using the instance of one of the concrete implementor. Now we are going to see a real-life example that we're going to implement in the next video and I hope that that example makes clear what this pattern is all about.

A screenshot of a bridge

Description automatically generated

1. Bridge – Hands-on Example UML:

Finally, the example that I have been promising you that will make things clear to you. Okay, So here we have an example which is using Bridge design pattern. So we have a Client, it can be any code, Java code. And this Client needs a collection which provides first in, first out type of algorithm. So we have a interface or abstract class here called as FifoCollection that has operations defined as offer() and poll(). So offer() will basically take an object and add it into collection(i.e., FifoCollection) and poll() will return the object based upon first in, first out principle. Then we have a refined abstraction. So we have a Queue here that implements the FifoCollection operations and it provides the implementation for offer() and poll() methods. So our client is going to use the Queue class. On the right hand side, you will see we have Implementor, so we have a class called as Linkedlist(it could be an interface or an abstract class) that has defined methods like add element to the start of our linked list[i.e., addFirst()], remove element from the start of our linked list[i.e., removeFirst()], add element to the end of our linked list[i.e., addLast()] and remove the last element from our linked list[i.e., removeLast()]. So this is our Implementor(i.e., Linkedlist) and it has defined its own set of methods that do not correspond to offer() and poll() methods defined in FifoCollection. So we don't have any offer() method in LinkedList. But what we can do is we can provide implementation of offer() method or first in, first out algorithm using these methods[addFirst(), removeFirst(), addLast() and removeLast()]. And that's what Bridge is all about. At the bottom you will see we have two implementations for LinkedList. We have an ArrayLinkedList, so this is going to use an array in order to create the linked list. Then we have a SinglyLinkedList that is going to use Node class in order to create the linked list. Between left and right inheritance hierarchy, there is a bridge. So our FifoCollection(i.e., more specifically, Queue) instance is going to have an instance or object of LinkedList(i.e., more specifically, any class which implements this LinkedList). So this is how Bridge design pattern works. When we are talking about abstraction here(left inheritance hierarchy), we can add any new class to this particular inheritance hierarchy or we can modify our existing FifoCollection abstract class/interface, so we can add another method that says, maybe, peek() method which simply returns whatever is present at the top of Queue, but it won't remove it. Do we need any change in the LinkedList(right inheritance hierarchy)? No. So this is what they mean when they say that you can change the abstraction(left inheritance hierarchy) without needing change to implementation. So we can add a peek() method in our FifoCollection(left inheritance hierarchy) and we can provide implementation of that without requiring any change to our LinkedList OR we can change our LinkedList(right inheritance hierarchy) and provide a method that basically says clear everything[i.e., clear()] and we don't need any change in our abstraction(left inheritance hierarchy). So the key to understanding Bridge design pattern is that when they say that we can vary/change abstraction and implementation separately, they are talking about the abstraction in terms of methods that are used by client and these methods hide how the functionality is provided using the implemented classes. Okay, so I hope this clears and explains to you what Bridge design pattern is.



1. Bridge – Implementation:

It's time to see how we can implement the Bridge design pattern in Java. And as usual, I have written some of the classes that we are going to use so that we can save some time while implementing this design pattern. So these are the same classes that we saw in the UML diagram. So we have a FifoCollection here, that is a first in first out collection, a generic interface, and this is what our abstraction is. That means our Client is going to deal with this particular interface and implementation of this particular abstraction(i.e., FifoCollection). Then we have LinkedList, which is also an interface, but this is our implementor. That means LinkedList provides operations or defines operations(i.e., methods) that can be used by our abstraction(i.e., FifoCollection) to provide its functionality. Then we have a Queue which will be our concrete or refined abstraction. So we're going to implement this Queue class using one of the concrete implementation of LinkedList. To do that I have created here a SinglyLinkedList class which implements our LinkedList interface and provides implementation for methods defined in that interface(i.e., LinkedList). So let's start implementing Queue. Remember, this is our refined abstraction and as we saw our abstraction works or uses our implementor in order to provide its functionality. So I'm going to start by implementing FifoCollection interface. Now we saw that our abstraction uses composition and it uses an object of implementor. So for our LinkedList implementor, we have SinglyLinkedList which is a concrete class. So I'm going to have a private field to hold the LinkedList type of instance using which we can provide our functionality. So let's say we want to provide implementation for our offer() method inside the Queue class. To do that, we're going to simply say 'list.addLast(element)'.So we're going to add the element to end of our linked list. And when it comes to remove the 1st element from our linked list, we're going to say 'list.removeFirst()'. So this method returns the first element that is present in our linked list and removes it as well. So this is how we can provide the functionality in our abstraction using the implementor. Now, one thing that remains is we need to define a public constructor that can be used in order to get hold of our LinkedList type of instance. This is how a Bridge design pattern works. So this particular composition(private LinkedList<T> list) that we have here(inside Queue class) is what bridge is all about. Now how do we use it(Bridge design pattern)? So we are going to jump to the Client class and we are going to say, let's create a FifoCollection of, let's say integers. So, we write 'FifoCollection<Integer> collection = new Queue<>(new SinglyLinkedList<>());'. So now our 'collection' is ready to be used and we can use it and call the method that are defined there. So we can call the offer() method and add some integers(let's 10, 40, 99). Then we can check whether our 'collection' returns these values by simply printing it. So we wanted a first in first out collection, so we added 10 first and we want to get 10 back as the first element. So, I am going to call poll() method on 'collection' reference variable to remove and return the 1st element from the linked list(i.e., Queue). I'm going to call this method[i.e., poll()] three times. Let's call it[i.e., poll()] one more time to make sure that we have handled the null issues/problems. So, if the queue(implemented by linked list) is having no elements in it i.e., size = 0 then we are returning 'null'. And if we run this Client code using Run as Java Application in Eclipse, you will see that we get our first element that was added, then the second and then the last. Lastly, we get null. So whether to return 'null' or throw an exception is up to the implementation and that decision can be taken either in the abstraction or the implementer can throw an exception as well. Anyways, the point here is how we implemented our abstraction so that it uses composition and one of the concrete implementor to provide functionality that is defined here(inside abstraction). So this is how you can implement Bridge design pattern in Java.

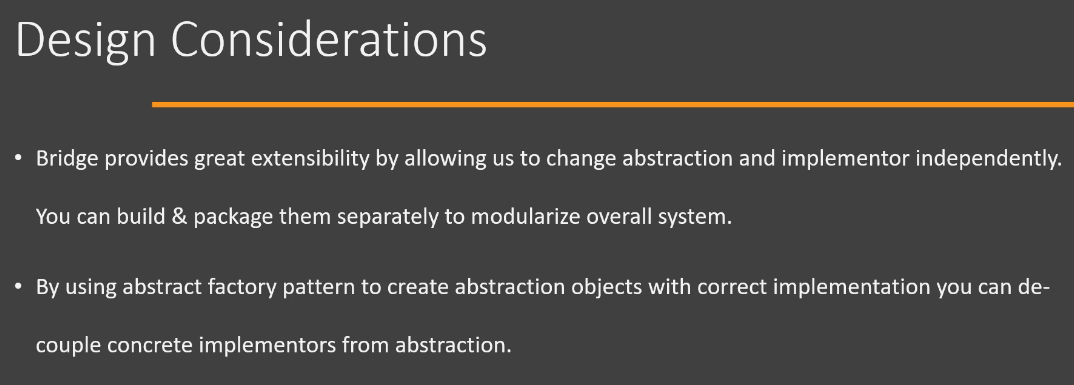
1. Bridge – Implementation & Design Considerations:

Now let's look at the implementation and design considerations when working with Bridge design pattern. So in case we are ever going to have a single implementor class in our application, then you can skip implementing the abstract implementor. This is not the requirement of this design pattern. Next, the abstraction which is going to use our implementor, it can create object of that single implementor on its own. So you can hardcode the class instantiation that it is going to use in the constructor of abstraction, create an object and use it. The second way, which is the preferred way, is we can give the concrete implementor object to our abstraction at runtime, but that makes sense if you have or in future if you are going to have multiple concrete implementor. Now, the last approach that we discussed where we give implementor object to the abstraction at runtime, it has a great potential in that, we can provide another implementor in future to our abstraction, and that way we can get a decoupling between the abstraction and implementor.

A black and white text on a black background

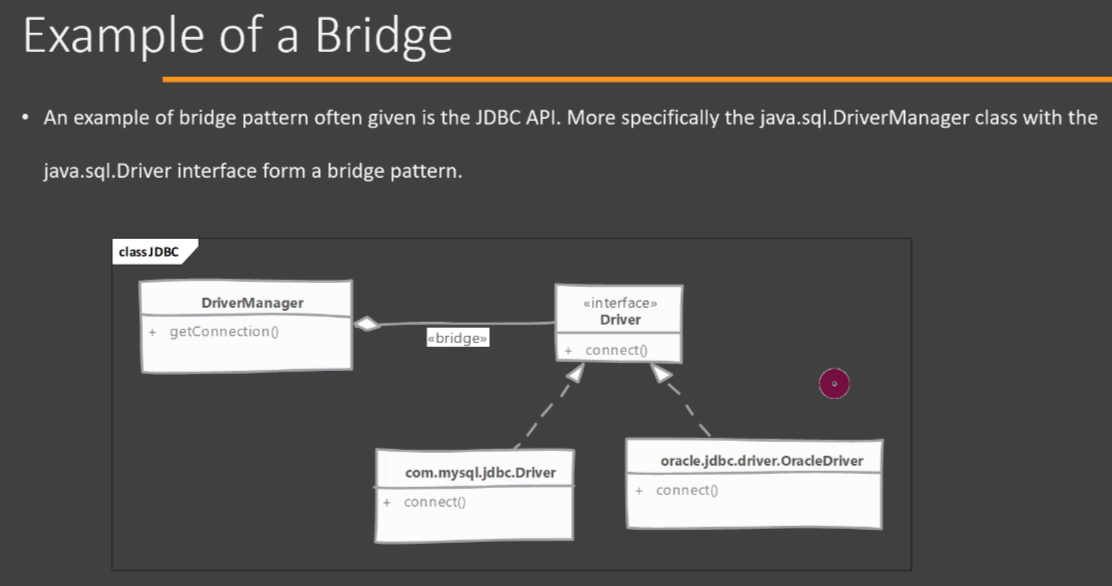
Description automatically generated

Now let's look at some of the design considerations when working with Bridge design pattern. Bridge provides a great flexibility and extensibility in allowing us to change both the abstraction and implementation/implementor. And the design point that I wanted to highlight here is that you can implement your abstraction and implementor separately and you can package them separately. So you will have a single jar which has only the abstraction that is used by Client and at runtime you can add another jar which is an implementor in the classpath, and the abstraction can using the newly added implementation. So there is a great benefit that you can have while development as well as while deploying by separating the abstraction and implementor. Next, if we use Abstract factory design pattern, then we can create our abstraction objects with the correct implementor that they are going to use and achieve a great decoupling between these two abstraction and implementor. So these are some of the points that you should consider when working with Bridge design pattern.



1. Bridge – Real World Example:

Now let's look at some examples of Bridge design pattern. Now, one great example of this design pattern is found in the JDBC API. More specifically, when we are dealing with DriverManager and the Driver interface that is exposed or that is defined in the JDBC API. So in our code(i.e., Client code), we will deal or we will work with DriverManager. So in your code(i.e., Client code), you will use the DriverManager class to call the getConnection() method and use JDBC API in order to interact with the database. On the other side of the bridge, we have the actual implementations of the Driver interface, which are specific to a particular driver vendor. So we have a different RDBMS and for each RDBMS, a set of classes is provided by the corresponding vendor. A set of classes for connecting to MySQL database, a different set of classes for connecting with Oracle. And, we can switch these implementations and our code can continue to work with our DriverManager. So we have these two separate parts of inheritance hierarchies, so our Driver can change without needing to change our Client which is working with DriverManager. On the other hand, if we change the DriverManager, for example, how it is defined, our Drivers don't need to modify. So, this is how the Bridge design pattern is used in the JDBC API.



Now another example of Bridge design pattern that is often given is the newSetFromMap() method, which is in the Collections class of java.util package. Now this method[i.e., newSetFromMap()] actually returns a set. The newSetFromMap() method returns a set which is backed by the Map object that we have given as an argument to this method. So let's look at the code of the actual class(i.e., SetFromMap) whose object is returned by this method[i.e., Collections.newSetFromMap()]. So you will see that this is a code that is taken from the Collections.class file and we have a class here called as SetFromMap. It is a private class, so it is not directly accessible to the outside world. So it is a SetFromMap class and you can see that in the constructor, we give the actual Map object and this class will actually work by using methods that are defined in the Map. So if the Map is changed, then the interface(here interface means not Java's interface. It means the way it is visible to customers/clients) of SetFromMap(Here, interface of SetFromMap means, method name of SetFromMap class) doesn't need to change. only the implementation i.e., delegation of method call, needs to be changed. So it(i.e., SetFromMap) can start using some different methods if our Map interface is changed. If we decide to make changes to the interface(it means how it is looked to the clients) that is exposed by this SetFromMap class, we don't need to make changes to the Map, we simply need to adjust the implementation of these methods present in SetFromMap class without worrying about the Map. All right, so this is another example of Bridge design pattern.

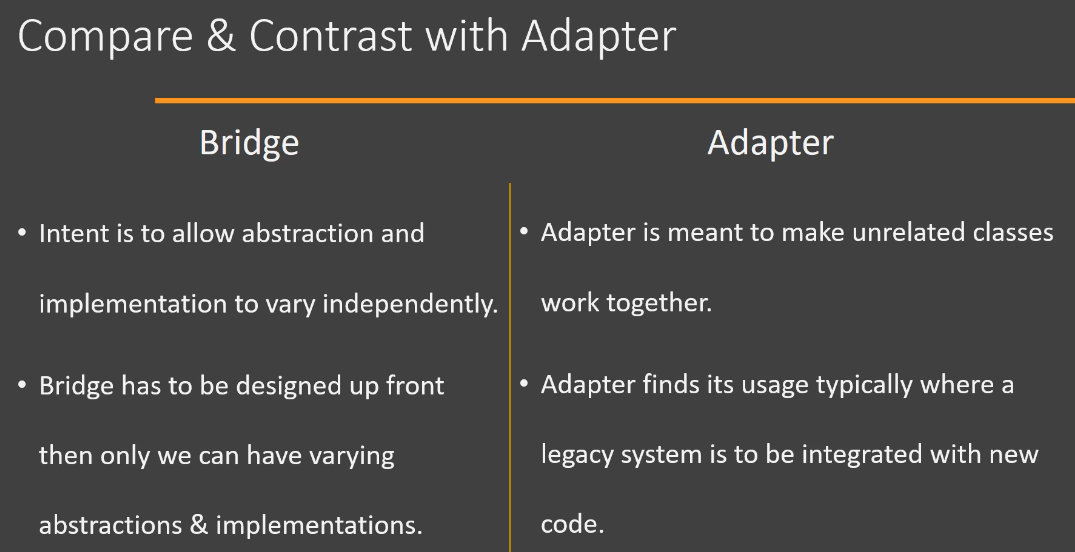
Now one thing that you will quickly pick up is that Bridge and Adapter design pattern are quite common. In fact, if you look at the structure of this SetFromMap class, the way they are implemented, you will find that they are quite similar. We, in the Adapter design pattern as well, especially in the Object Adapter, you will see that we also give an object to our Adapter's constructor and it provides some methods and delegates the method calls on the object that it was given. So Bridge design pattern also works on the similar principle but here we pass an object of the implementation which is used by our abstraction hierarchy. The difference between these two patterns, however, is the intent or purpose of using these two design patterns. Adapter is written when we already have an existing class and we already have a requirement of adapting this class to different object. Bridge, however, is written from the ground up. So when we are writing our SetFromMap class, we are making the decision that we are going to use an existing Map object. All right, so there is a difference of the purpose between the Bridge and Adapter design pattern. However, if you look at the code of SetFromMap class then it can be quite confusing to differentiate between Adapter and Bridge. So always remember that whenever you are thinking about design patterns, you should always think conceptually that what is the purpose of this particular design pattern, instead of going into how this pattern is implemented, that way you will avoid a lot of confusion while differentiating between these patterns.

A screenshot of a computer program

Description automatically generated

1. Bridge – Comparison with Adapter:

Now let's look at the differences between Bridge and Adapter design pattern. So we saw/know what Adapter design pattern is, and it has the Object Adapter implementation which also needs object of another class in it, and it is using that object to provide the functionality. So it's important that you understand what the difference between Bridge and Adapter is, because in Bridge as well, we have an abstraction which contains the object of implementor. So first difference is that intent behind these design patterns. Bridge has a purpose of allowing the abstraction and implementation to vary independently. So that's what the Bridge pattern is used for. Adapter, on the other hand, is meant to make unrelated classes work together. So we have a client class and then we have our Adaptee. And by implementing the Target interface and using the Adaptee object, Adapter makes these classes work together. So our Client works with our Adaptee. So that is the purpose behind Adapter design pattern. Next, one point that you will notice from the UML that we saw about Bridge is that this design pattern needs to be designed and implemented from the get-go(i.e., from the beginning). Once you have all these classes in place, then it's quite difficult to refactor them in the Bridge design pattern. Adapter, on the other hand, by definition, is used whenever we already have an existing object. Okay, so it's very useful if you are integrating a legacy system into your new system. So you can have an Adapter that uses object from legacy system and implements interface that is used by our brand-new system. So this is the difference between Bridge and Adapter design pattern.



1. Bridge – Pitfalls:

Now let's look at pitfalls of Bridge design pattern. If you are learning about this design pattern for the very first time, then you can agree that it's fairly complex to understand what exactly Bridge does. And it's quite complex to implement as well. So that is one of the pitfalls of the Bridge design pattern. Second pitfall is that the decision to use Bridge design pattern, in order to make that decision correctly, you need to have in front of you the entire design. Only then, after looking at all the classes and how they are interacting with each other, you can take a decision that you can implement Bridge design pattern. And that brings us to the last point, which again says that it's quite difficult to implement Bridge design pattern if your implementation is already completed, because it changes how the classes or objects interact with each other. And even if you have an ongoing project, adding this design pattern at a later stage of development still may require a lot of rework and another round of unit testing. So these are some of the pitfalls of using Bridge design pattern.

A screenshot of a computer

Description automatically generated

1. Bridge – Summary:

So here is a summary of Bridge design pattern. We use this design pattern whenever we want our abstraction, that is the interface(here interface means the way it is looked to the client), that is used by our client and the implementors, these are the classes which provide this functionality, we want to separate or decouple them. Bridge achieves this decoupling by defining separate inheritance hierarchies for our abstraction(these are the classes that are used by our client code) and the implementors which provide the actual functionality. So we have these two separate inheritance hierarchies, and we use composition in order to connect these two hierarchies. And this connection using composition is what gives this design pattern its name called as Bridge. Now, the implementors do not have to define the exact same methods that are defined in abstraction. What is common practice or what you will typically find in real life is that implementors have these primitive methods, methods that do small amount of work, and then we have abstractions which use these small methods in order to provide a functionality which corresponds to a use case that our client code wants to use.

